

Department I - C Plus Plus

Modern and Lucid C++  
for Professional Programmers

Week 7 – Standard Containers & Iterators

Prof. Peter Sommerlad / Thomas Corbat  
Rapperswil, 29.10.2019  
HS2019



## Recap Week 6



- Namespaces group names
- The same namespace can be opened multiple times
- Defining a member of a namespace requires the namespace name as a qualifier
- The anonymous namespace hides names

```
namespace demo {  
void foo(); //1  
namespace subdemo {  
void foo() { /*2*/ }  
} // subdemo  
} // demo  
  
namespace demo {  
void bar() {  
    foo(); //1  
    subdemo::foo(); //2  
}  
}  
  
void demo::foo() { /*1*/ } // definition  
  
int main() {  
    using demo::subdemo::foo;  
    foo(); //2  
    demo::foo(); //1  
    demo::bar();  
}
```

- **Types and (non-member) functions belonging to that type should be placed in a common namespace**

```
namespace calendar { ... }
```

- **Advantage: Argument Dependent Lookup!**
  - When the compiler encounters an unqualified function or operator call with an argument of a user-defined type it looks into the namespace in which that type is defined to resolve the function/operator
  - E.g. it is not necessary to write `std::` in front of `for_each` when `std::vector::begin()` is an argument of the function

```
namespace calendar {                                     date.h  
  
class Date {  
    //...  
};  
  
bool is_holiday(Date const &);  
  
}
```

```
#include "date.h"                                       calendar.cpp  
  
using Dates = std::vector<calendar::Date>;  
  
void mark_holidays(Dates const & dates) {  
    for_each(begin(dates), end(dates),  
        [](calendar::Date const & d) {  
            if (is_holiday(d)) //calendar::is_holiday  
                //...  
        });  
}
```

- Enumerations are useful to represent types with only a few values

```
enum <name> {  
    <enumerators>  
};
```

```
enum class <name> {  
    <enumerators>  
};
```

- An enumeration creates a new type that can easily be converted to an integral type (for unscoped enumeration)
  - Conversion from an integral type to an enumeration is not possible implicitly
- The individual values (enumerators) are specified in the type
- Unless specified explicitly the values start with 0 and increase by 1

- Unscoped enumeration (no class keyword)

```
enum DayOfWeek {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
}; 0    1    2    3    4    5    6
```

- Enumerators leak into surrounding scope
- Best used as member of a class

- Scoped enumeration (class keyword)

```
enum class DayOfWeek {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
}; 0    1    2    3    4    5    6
```

- Enumerators do not leak into surrounding scope
- Avoids implicit conversion to int

# STL Containers: General API

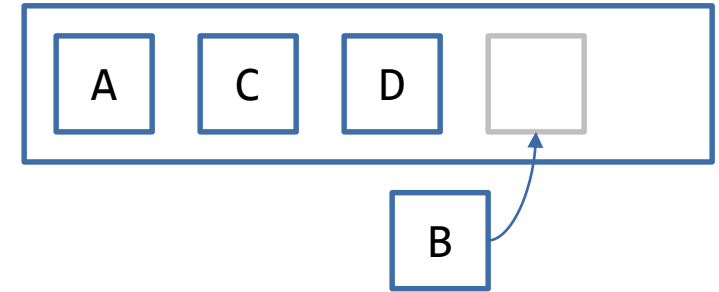


## Goals:

- You know the different standard containers categories
- You know the properties of the categories
- You know common functionality of most standard containers

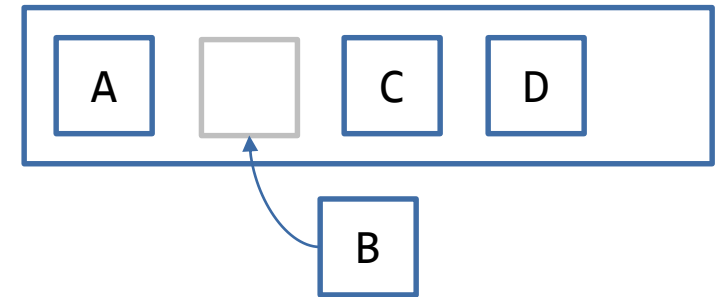
- **Sequence Containers**

- Elements are accessible in order as they were inserted/created
- Find in linear time through the algorithm find



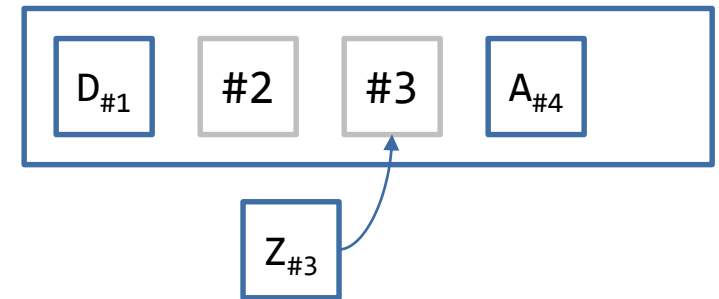
- **Associative Containers**

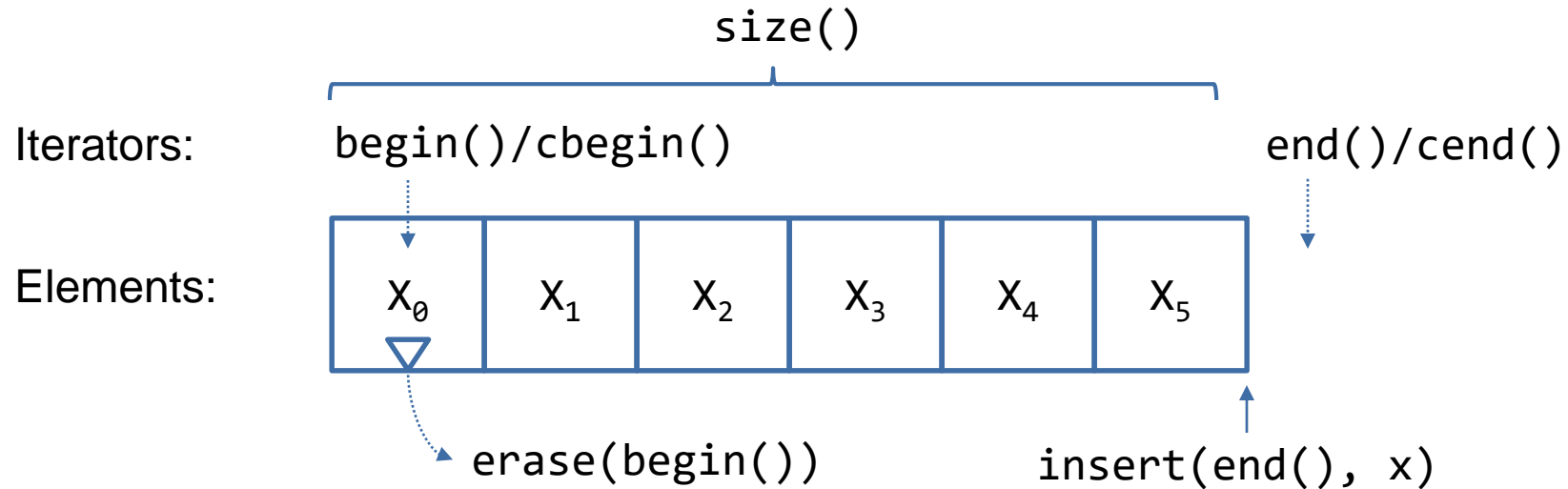
- Elements are accessible in sorted order
- find as member function in logarithmic time



- **Hashed Containers (Unordered Associative)**

- Elements are accessible in unspecified order
- find as member function in constant time





- All containers have the same/a similar basic interface

Member Function	Purpose
<code>begin()</code> <code>end()</code>	Get iterators for algorithms and iteration in general
<code>erase(iter)</code>	Removes the element at position the iterator <code>iter</code> points to
<code>insert(iter, value)</code>	Inserts <code>value</code> at the position the iterator <code>iter</code> points to
<code>size()</code> <code>empty()</code>	Check the size of the container



- **Containers can be...**

- ... default-constructed
- ... copy-constructed from another container of the same type
- ... equality compared if they are of the same type  
(or even lexicographically compared with relational operators)  
as long as their elements can be compared accordingly
- ... emptied with `clear()`

```
std::vector<int> v{};  
std::vector<int> vv{v};  
if (v == vv) {  
    v.clear();  
}
```

- **Construction with initializer list**

```
std::vector<int> v{1, 2, 3, 5, 7, 11};
```

v: 

1	2	3	5	7	11
---	---	---	---	---	----

- **Construction with a number of elements**

- Can provide default value
- Often needs parenthesis instead of {} to avoid ambiguity from list of values initialization

```
std::list<int> l(5, 42);
```

l: 

42	↔	42	↔	42	↔	42	↔	42
----	---	----	---	----	---	----	---	----

- **Construction from a range given by a pair of iterators**

- might need parenthesis instead of {} (rare)

```
std::deque<int> q{begin(v), end(v)};
```

q: 

1	2	3	5	7	11
---	---	---	---	---	----

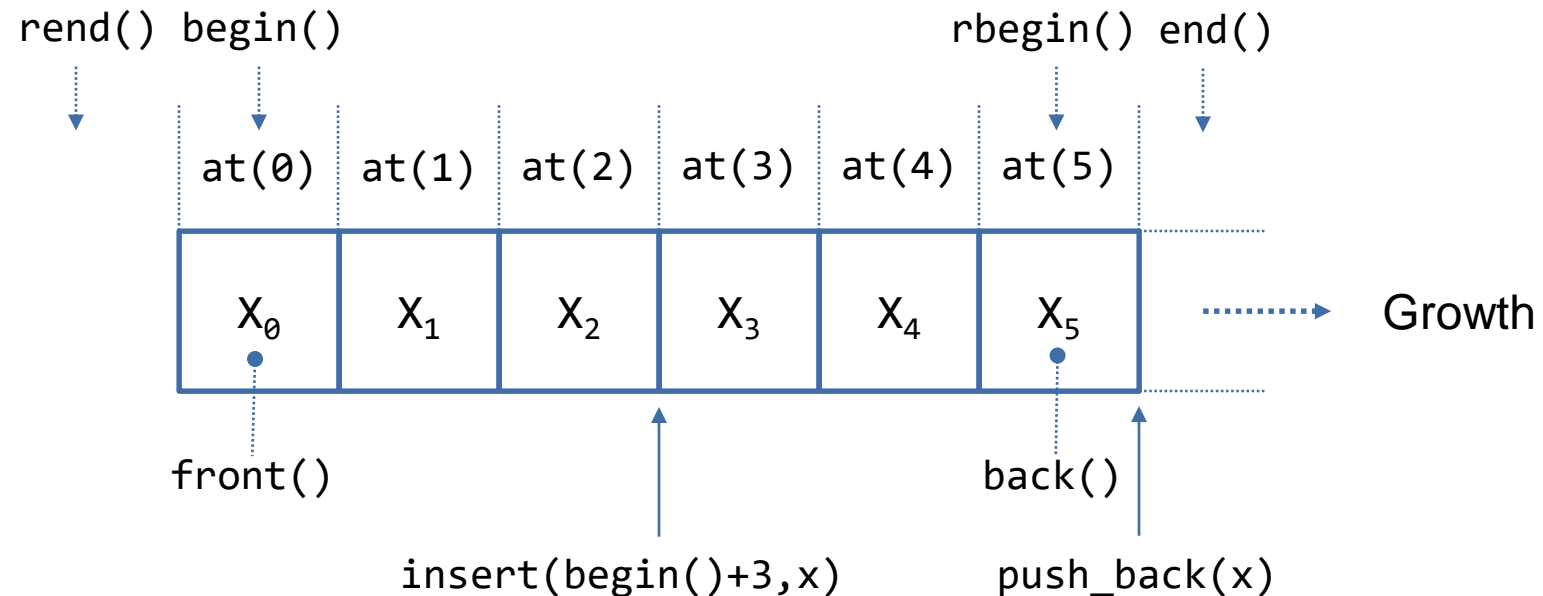
# Sequence Containers



## Goals:

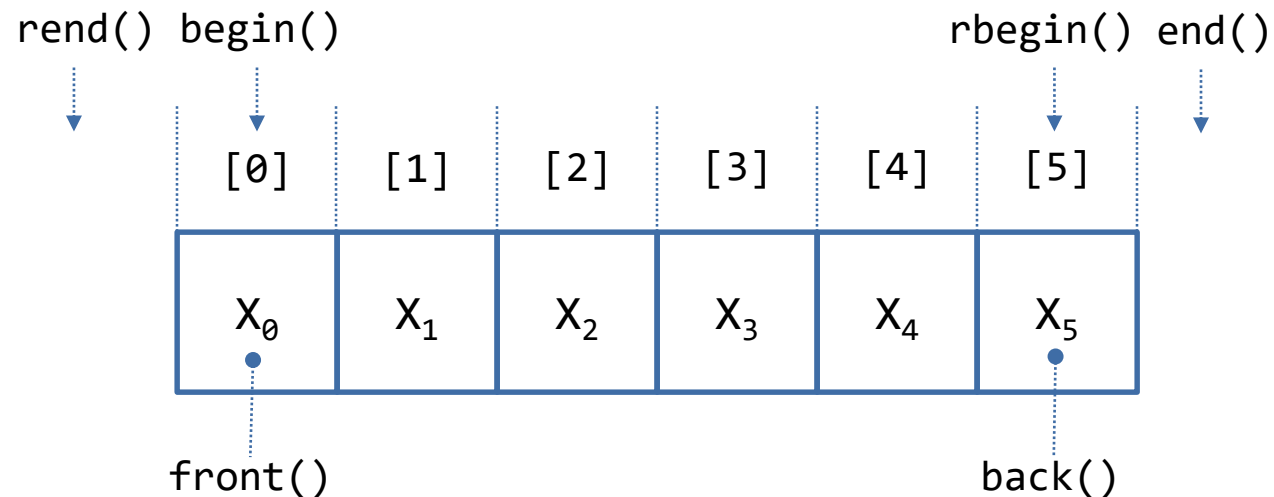
- You know the sequence containers of the standard library
- You know the capabilities of the individual sequence containers

```
std::vector<T>  std::deque<T>  std::list<T>  std::forward_list<T>
```



- Define order of elements as inserted/appended
- Lists are good for splicing and in the middle insertions
- `std::vector`/`std::deque` are efficient unless bad usage

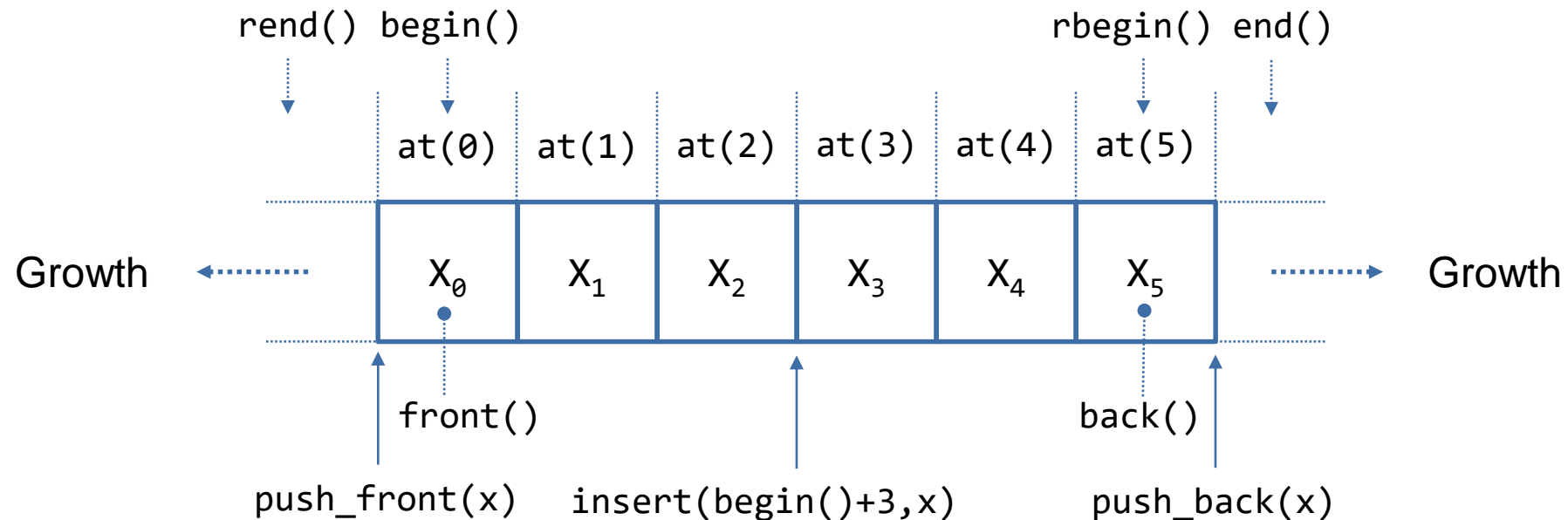
```
std::array<int, 6> values{1, 1, 2, 3, 5, 8};
```



- Fixed-size "container", cannot insert/append
- Can be initialized at compile-time
- Use `std::array` instead of C-style arrays when defining arrays

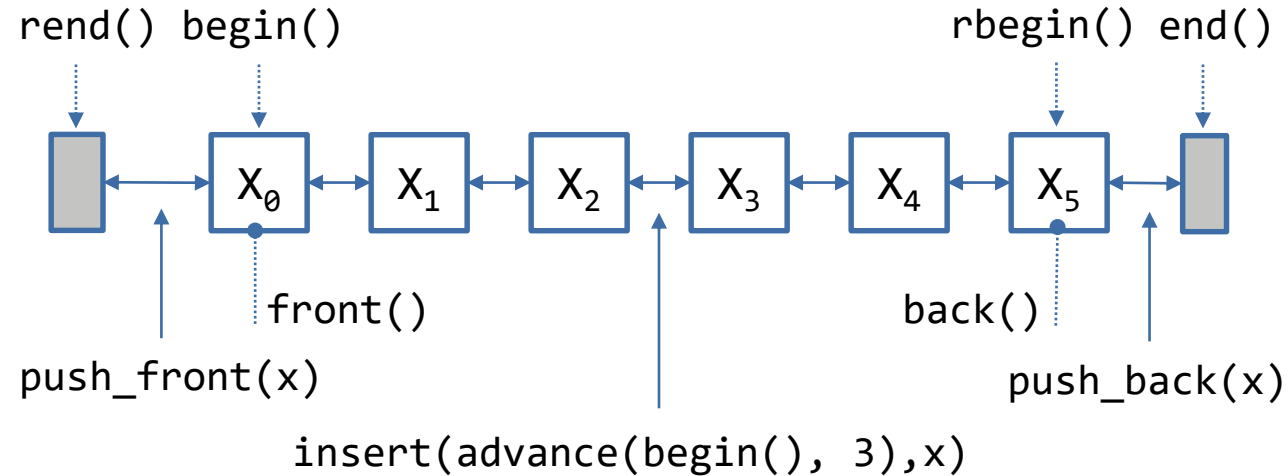
```
int obsolete[]{1, 1, 2, 3, 5, 8};
```

```
std::deque<int> q{begin(v), end(v)};  
q.push_front(42);  
q.pop_back();
```



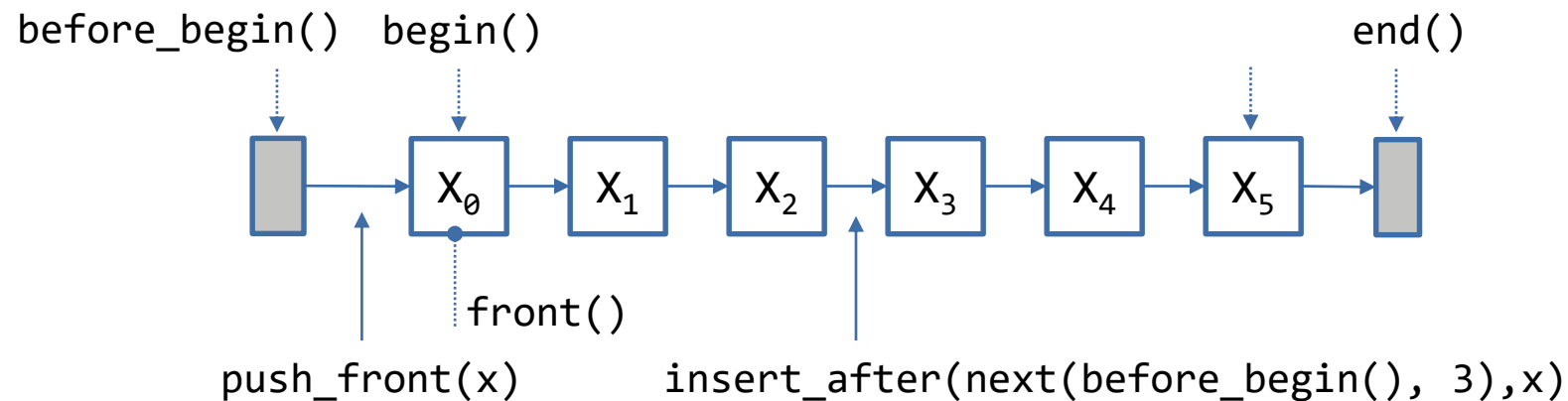
- **`std::deque` is like `std::vector` but with additional, efficient front insertion/removal**
  - `push_front()` and `pop_front()`
- **Special implementation for `bool`: `std::vector<bool>` and `std::deque<bool>`**

```
std::list<int> l(5, 1);
```



- Efficient insertion in any position
- Lower efficiency in bulk operations
- Requires member-function call for `sort()` etc.
- Only bi-directional iterators – no index access!

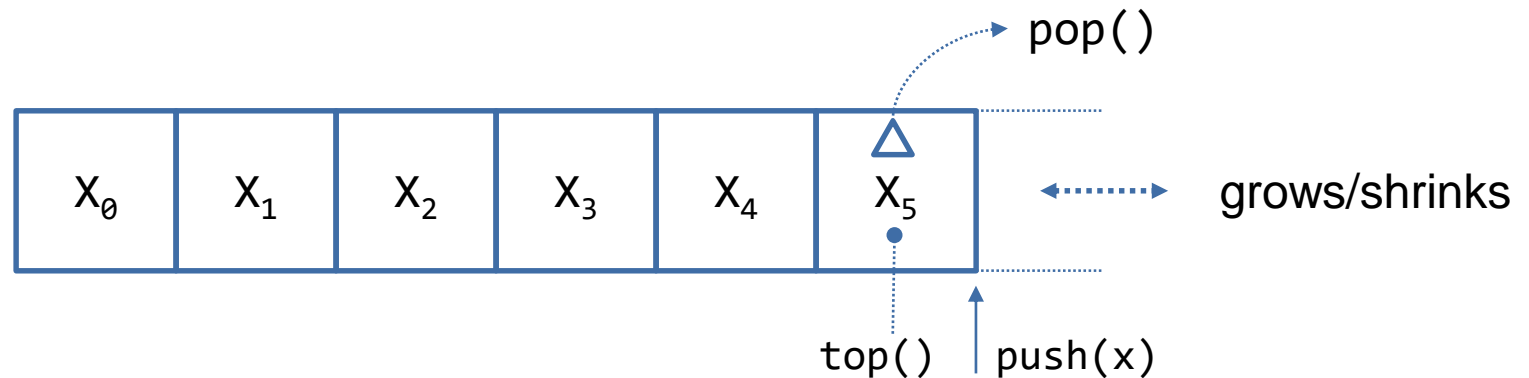
```
std::forward_list<int> l{1, 2, 3, 4, 5, 6};
```



- Efficient insertion AFTER any position, but clumsy with iterator to get "before" position
- Only forward-iterators, clumsy to search and remove, use member-functions not algorithms
- Avoid, except when there is a specific need! Better use `std::list` or even better `std::vector`

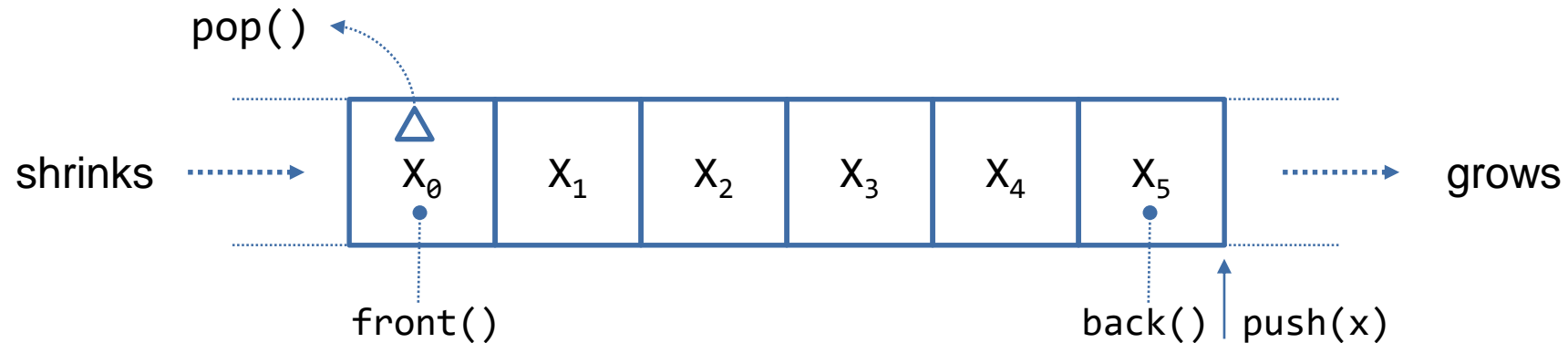


```
std::stack<int> s{};  
s.push(42);  
std::cout << s.top();  
s.pop();
```

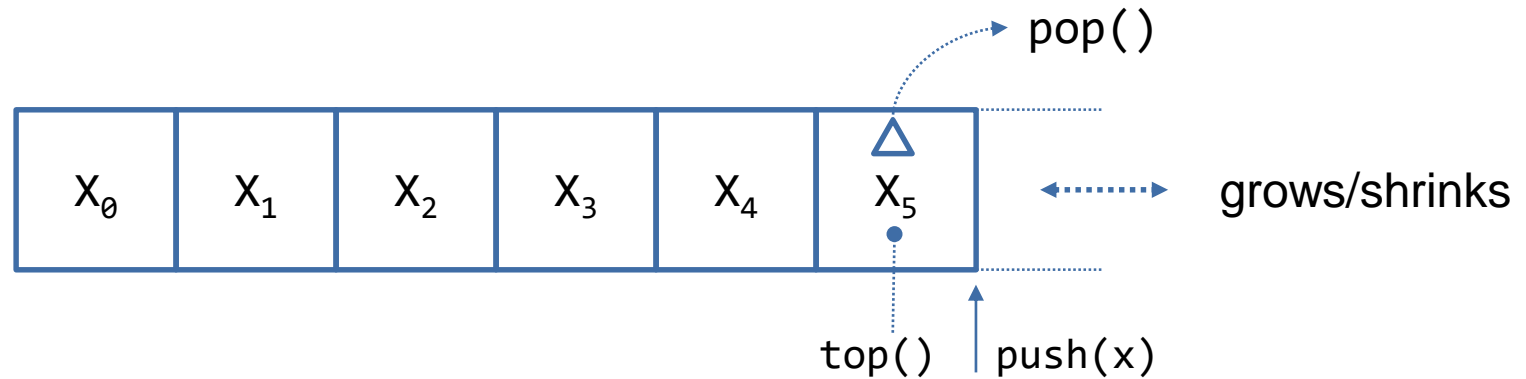


- **Uses `std::deque` (or `std::vector`, `std::list`) and limits its functionality to stack operations**
  - Delegates to `push_back()`, `back()` and `pop_back()`
  - Iteration not possible
- **No longer a container, deliberate limitation!**

```
std::queue<int> q{};  
q.push(42);  
std::cout << q.front();  
q.pop();
```



- **Uses `std::deque` (or `std::list`) and limits its functionality to queue operations**
  - Delegates to `push_back()` and `pop_front()`
  - Iteration not possible
- **No longer a container, deliberate limitation!**



- **Uses `std::deque` (or `std::vector`) and limits its functionality to stack operations**
  - But keeps elements partially sorted as (binary) heap
- **`top()` element is always the smallest (requires element type to be comparable)**
- **No longer a container, deliberate limitation!**

```
#include <stack>
#include <queue>
#include <iostream>
#include <string>

int main() {
    std::stack<std::string> lifo{};
    std::queue<std::string> fifo{};
    for (std::string s : { "Fall", "leaves", "after", "leaves", "fall" }) {
        lifo.push(s);
        fifo.push(s);
    }
    while (!lifo.empty()) { // fall leaves after leaves Fall
        std::cout << lifo.top() << ' ';
        lifo.pop();
    }
    std::cout << '\n';
    while (!fifo.empty()) { // Fall leaves after leaves fall
        std::cout << fifo.front() << ' ';
        fifo.pop();
    }
}
```

```
#include <algorithm>
#include <list>
#include <stdexcept>

int median(std::list<int> & values) {
    if (values.empty()) {
        throw std::invalid_argument{"empty..."};
    }
    sort(begin(values), end(values));
    return values[values.size() / 2];
}
```

Incorrect

std::list provides its own sort member function because it does not have random access iterators (see later in this lecture)

values.sort();

std::list does not provide index access operators

```
#include <queue>
#include <string>

using SplitFlapDisplay =
    std::queue<std::string>;

void flip(SplitFlapDisplay & display) {
    display.push(display.front());
    display.pop();
}
```

Correct

The std::queue features push and pop for modification. However, pop does not return the popped element. First it has to be queried with front.

This implies a copy. In C++ Advanced we will look at how to make this more efficient.

# Associative Containers



## Goals:

- You know the associative containers of the standard library
- You know the capabilities of the individual associative containers

- **Allow searching by content, not by sequence**
  - Search by key
  - Access key or key-value pair
- **Better name: "Sorted Associative Containers"**
- **Properties**

	Key Only	Key-Value Pair
Key Unique	<code>std::set&lt;T&gt;</code>	<code>std::map&lt;K, V&gt;</code>
Multiple Equivalent Keys	<code>std::multiset&lt;T&gt;</code>	<code>std::multimap&lt;K, V&gt;</code>

```
std::set<int> values{7, 1, 4, 3, 2, 5, 6};
```

- **Stores elements in sorted order (ascending by default)**

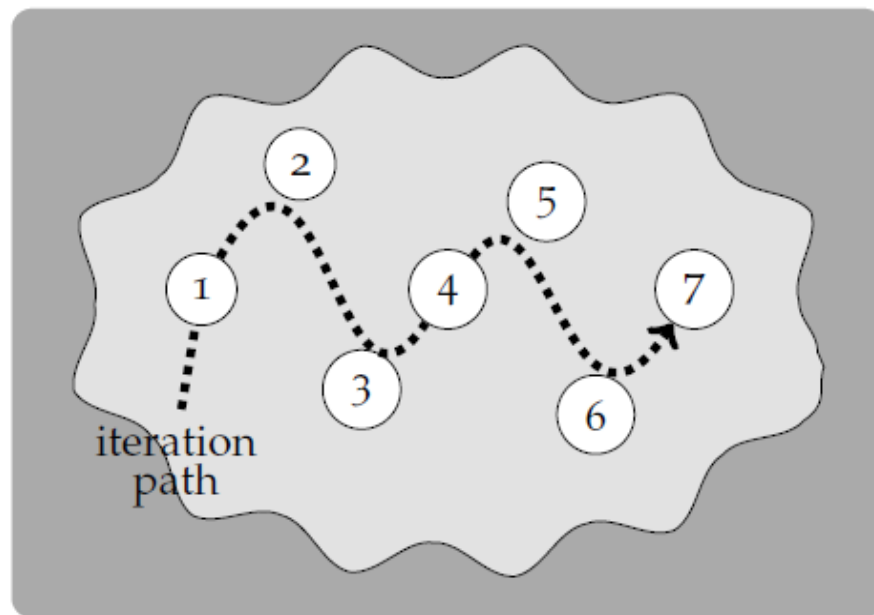
- Order can be overwritten by the 2nd template parameter

- **Iteration walks over the elements in order**

- Keys cannot be modified through iterators!

- **Use member functions for `.find` and `.count`**

- Tree-search instead of sequential search
- Result of `.count(element)` is either 0 or 1
- There is no `.contains(element)` member





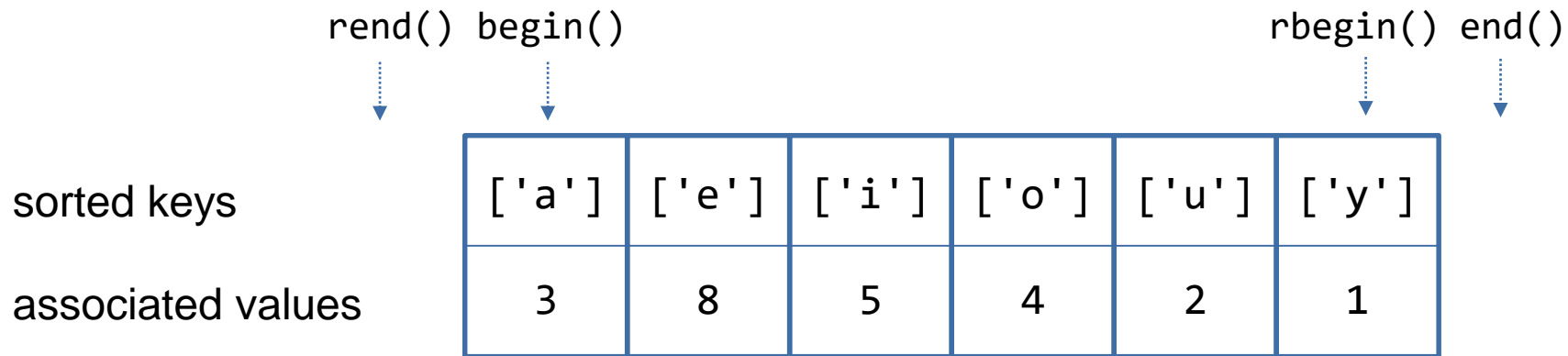
```
#include <set>
#include <iostream>

void filterVowels(std::istream & in, std::ostream & out) {
    std::set<const char> vowels{'a', 'e', 'o', 'u', 'i', 'y'};
    char c{};
    while (in >> c) {
        if (!vowels.count(c)) {
            out << c;
        }
    }
}

int main() {
    filterVowels(std::cin, std::cout);
}
```

- **Initializer does not need to be sorted**
- **`s.count(x)` as quick check if `x` is present in `std::set`**
  - Discouraged alternative (more code): `s.find(x) != s.end()`

```
std::map<char, size_t> vowels{{'a', 3}, {'e', 8}, {'i', 5}, {'o', 4}, {'u', 2}, {'y', 1}};
```



- **Stores key-value pairs in sorted order**
  - Sorted by key in ascending order
  - Order can be overwritten by the 3rd template parameter
- **Iterators access std::pair<key, value>**
  - Use .first for key and .second for value

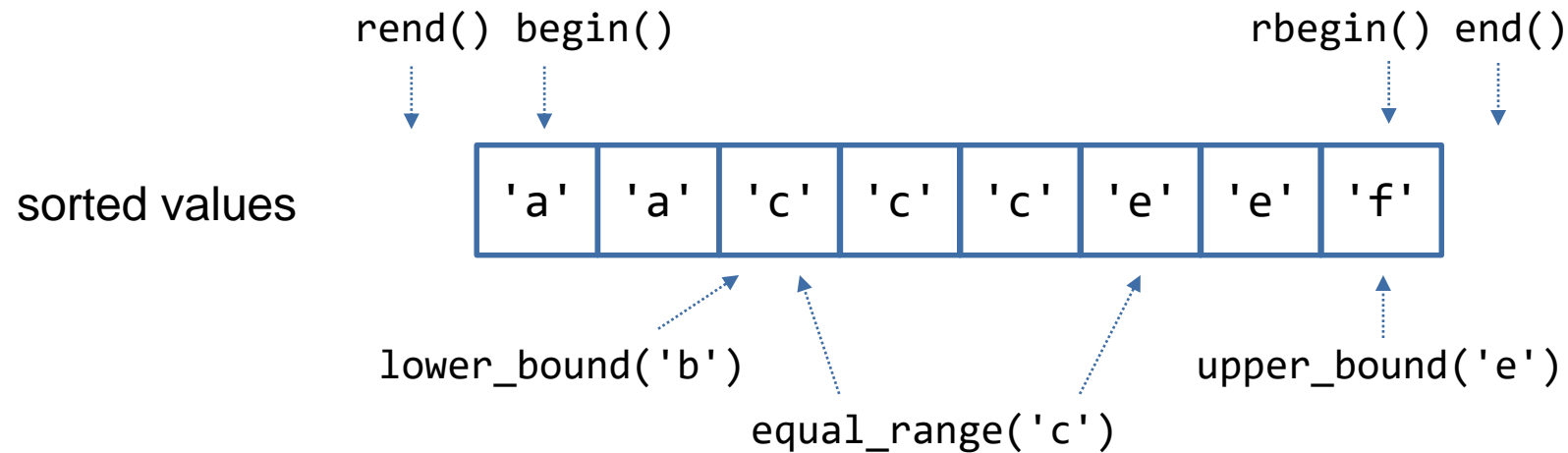
```
void countVowels(std::istream &in, std::ostream &out) {
    std::map<char, size_t> vowels{{'a', 0}, {'e', 0}, {'i', 0}, {'o', 0}, {'u', 0}, {'y', 0}};
    char c{};
    while (in >> c) {
        if (vowels.count(c)) { // only count those chars that are already in the map
            ++vowels[c];
            for_each(cbegin(vowels), cend(vowels), [&out](auto const & entry) {
                // entry is a pair<char, size_t>
                ut << entry.first << " = " << entry.second << '\n';
            });
        }
    }
}
```

- `m.count(x)` as quick check if `x` is a key present in `std::map`
  - Discouraged alternative (more code): `m.find(x) != m.end()`
- `auto` is simpler than `std::pair<key, value>`

```
void countStrings(std::istream & in, std::ostream & out) {
    std::map<std::string, size_t> occurrences{};
    std::istream_iterator<std::string> inputBegin{in};
    std::istream_iterator<std::string> inputEnd{};
    for_each(inputBegin, inputEnd, [&occurrences](auto const & str) {
        ++occurrences[str];
    });
    for(auto const & occurrence : occurrences) {
        out << occurrence.first << " = " << occurrence.second << '\n';
    }
}
```

- **Indexing operator[ ] inserts a new entry automatically if key is not present**
  - Key is the argument of the index operator, values is the default value of the value type

```
std::multiset<char> letters{'a', 'a', 'c', 'c', 'c', 'e', 'e', 'f'};
```



- **Multiple equivalent keys allowed**

- Use `equal_range()` or `lower_bound()/upper_bound()` member functions/algorithms to find boundaries of equivalent keys

- **Can be a bit more tedious to work with than `std::set`**

```
void sortedStringList(std::istream & in, std::ostream & out) {
    using inIter = std::istream_iterator<std::string>;
    using outIter = std::ostream_iterator<std::string>;
    std::multiset<std::string> words{inIter{in}, inIter{}};
    copy(cbegin(words), cend(words), outIter(out, "\n"));
    auto current = cbegin(words);
    while (current != cend(words)) {
        auto endOfRange = words.upper_bound(*current);
        copy(current, endOfRange, outIter{out, ", "});
        out << '\n'; // next range on new line
        current = endOfRange;
    }
}
```

- First copy-algorithm call prints each word on a single line
- Code in while-loop groups equivalent words on one line

```
#include <map>
#include <string>

std::map<double, std::string> const names {
    { 0.0, "zero"},
    { 1.0 / 0.0, "inifinity"},
    {-1.0 / 0.0, "-inifinity"},
    { 0.0 / 0.0, "NaN"}
};
```

Incorrect

The NaN value breaks the ordering of the keys, because all comparison operations return false.

```
#include <algorithm>
#include <set>

void incrementAll(std::set<int> & values) {
    for_each(begin(values), end(values),
        [](int & element) {
            ++element;
        });
}
```

Incorrect

Elements in an `std::set` (or keys in an `std::map`) must not be modified from outside. This might break the invariant that the elements are ordered. There is no means of recognizing such a modification when an element was modified through an iterator (or any other reference of that element)

# Hashed Containers



## Goals:

- You know the different hashed containers of the standard library
- You know the capabilities of the individual hashed containers



- **C++11 introduced associative containers using hashing**
  - More efficient lookup
  - No sorting
- **Standard lacks feature for creating your own hash functions**
  - `std::hash<T>` functor for library types and built-in types provided, esp. `std::string`
- **DIY programming of hash functions is hard and prone to failure, i.e. it might produce too many collisions -> stick to standard types, like `std::string` for keys**

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <unordered_set>

int main() {
    std::unordered_set<char> const vowels{'a', 'e', 'i', 'o', 'u'};
    using in = std::istreambuf_iterator<char>;
    using out = std::ostreambuf_iterator<char>;
    remove_copy_if(in{std::cin}, in{}, out{std::cout},
        [&](char c) { return vowels.count(c); }
    );
}
```

- **Usage is almost equivalent to `std::set`**
  - Except for the lack of ordering
- **Don't use `std::unordered_set` with your own types, unless you are an expert in hash functions and you benefit from the speedup**
  - Boost library provides hash-combiner helper

```
#include <unordered_map>
#include <iostream>
#include <string>

int main(){
    std::unordered_map<std::string, int> words{};
    std::string s{};
    while (std::cin >> s) { ++words[s]; }
    for(auto const & p : words) {
        std::cout << p.first << " = " << p.second << '\n';
    }
}
```

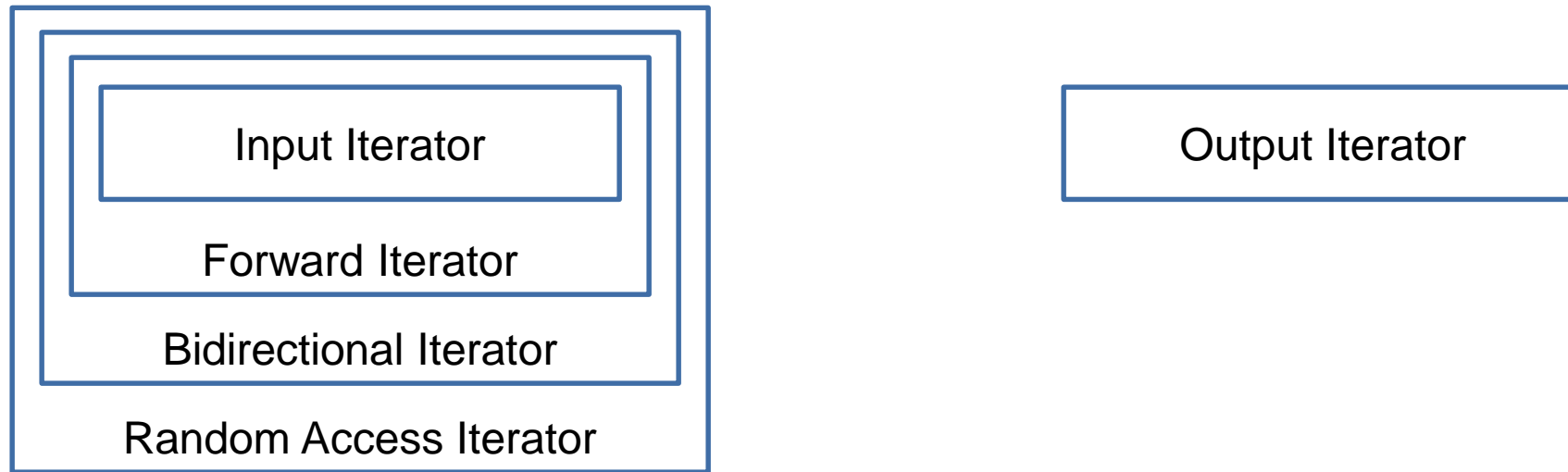
- **Usage is almost equivalent to `std::map`**
  - Except for the lack of ordering
- **Don't use `std::unordered_map` with your own types, unless you are an expert in hash functions and you benefit from the speedup**
  - Boost library provides hash-combiner helper

# Iterators



## Goals:

- You know the different iterator categories and their capabilities
- You can explain the difference between a const iterator and a const\_iterator



- **Different containers support iterators of different capabilities**
- **Categories are formed around increasing "power"**
  - `std::input_iterator` corresponds to `istream_iterator`'s capabilities
  - `std::ostream_iterator` is an `output_iterator`
  - `std::vector<T>` provides `random_access` iterators

- Supports reading the "current" element (of type `Element`)
- Allows for one-pass input algorithms
  - Cannot step backwards
- Models the `std::istream_iterator` and `std::istream`
- Can be compared with `==` and `!=`
  - To other iterator objects of the same type: `It`
- Can be copied
  - After increment (calling `++`) all other copies are invalid!
  - `*it++` is allowed explicitly (by the standard)

```
struct input_iterator_tag{};

Element operator*();
It & operator++();
It operator++(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor
```

- **Can do whatever an input iterator can, plus...**
  - Supports changing the "current" element (of type Element)
    - Unless the container or its elements are const
- **Still allows only for one-pass input algorithms**
  - Cannot step backwards
  - But can keep iterator copy around for later reference
- **Models the `std::forward_list` iterators**

```
struct forward_iterator_tag{};

Element & operator*();
It & operator++();
It operator++(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor
```

- Can do whatever a forward iterator can, plus...
  - Can go backwards
- Allows for forward-backward-pass algorithms
- Models the `std::set` iterators

```
struct bidirectional_iterator_tag{};

Element & operator*();
It & operator++();
It operator++(int);
It & operator--();
It operator--(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor
```



- **Can do whatever a bidirectional iterator can, plus...**
  - Directly access element at index (offset to current position): distance can be positive or negative
  - Go n steps forward or backward
  - "Subtract" two iterators to get the distance
  - Compare with relational operators (<, <=, >, >=)
- **Allows random access in algorithms**
- **Models the `std::vector` iterators**

```
struct random_access_iterator_tag{};

Element & operator*();
It & operator++();
It operator++(int);
It & operator--();
It operator--(int);
bool operator==(It const &);
bool operator!=(It const &);
It & operator=(It const &);
It(It const &); //copy ctor

Element & operator[](distance);
It operator+(distance);
It & operator+=(distance);
It operator-(distance);
It & operator-=(distance);
distance operator-(It const &);
//relational operators, like <
```

- **Can write value to current element, but only once**  
(`*it = value`)
  - Then increment is required
- **Modeled after `std::ostream_iterator`**
- **Most other iterators can also act as output iterators**
  - Unless the underlying container is `const`
- **Exception: associative containers allow only read-only iteration**
- **No comparison and end to an out range is not queryable**

```
struct output_iterator_tag{};

Element & operator*();
It & operator++();
It operator++(int);
```

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

- **Some algorithms only work with powerful iterators, e.g. `std::sort()` requires a pair of random access iterators (it needs to jump forward and backward)**
- **Some algorithms can be implemented better with more powerful iterators**  
E.g. `std::advance()` or `std::distance()`
- **Categories will be enforcable with C++ concepts (probably in 2020)**

```
std::distance(start, goal);  
std::advance(itr, n);
```

- **std::distance()** counts the number of "hops" iterator start must make until it reaches goal
  - Efficient for random access iterators
  - For other iterators the algorithm has to loop
    - This implies that goal has to be "after" start. I.e. reachable by an arbitrary number of ++ calls
- **std::advance()** lets itr "hop" n times
  - Efficient for random access iterators
  - For other iterators the algorithm it has to loop
  - Allows negative n for bidirectional iterators

```
int main() {  
    std::vector<int> primes{2, 3, 5, 7, 11, 13};  
  
    auto current = std::begin(primes);  
    auto afterNext = std::next(current);  
    std::cout << "current: " << *current << " afterNext: " << *afterNext << '\n';  
  
    std::advance(current, 1);  
    std::cout << "current: " << *current << " afterNext: " << *afterNext << '\n';  
}
```

### ● std::next / std::prev

- Has a default step of size 1, can be specified
- Makes a copy of the argument
- Argument can be a temporary

### ● std::advance

- Requires a step
- Modifies the argument iterator
- Returns void

```
std::vector<int> v{3, 1, 4, 1, 5, 9, 2, 6};  
for (auto it = cbegin(v); it != cend(v); ++it) {  
    std::cout << *it << " is " << ((*it % 2) ? "odd" : "even") << '\n';  
}
```

- **Use auto**

- because begin()'s return type is often long (pre C++11 see below)

- **Use begin() and end() if you intend to change elements**

- Otherwise cbegin() and cend()

```
for (std::vector<int>::const_reverse_iterator rit = crbegin(v); rit != crend(v); ++rit) {  
    std::cout << *rit << ", ";  
}
```

- **Declaring an iterator const would not allow modifying the iterator object**
  - You cannot call ++
- **cbegin() and cend() return const\_iterators**
  - This does NOT imply the iterator to be const
  - The elements the iterator walks over are const

```
std::vector<int> v{3, 1, 4, 1, 5, 9, 2, 6};
```

```
auto const iter1 = values.begin(); //std::vector<int>::iterator const  
++iter1;
```

```
auto iter2 = values.cbegin(); //std::vector<int>::const_iterator  
*iter = 2;
```

```
using InIter = std::istreambuf_iterator<char>;
std::optional<char> last (std::istream & in) {
    InIter current{in}, eof{}, previous{};
    while (current != eof) {
        previous = current++;
    };
    if (previous != eof) {
        return *previous;
    }
    return{};
}
```

### Incorrect

Beware! This compiles and might work as expected, but is not guaranteed! However, after an input iterator has been incremented, its copies are invalidated!

```
int median(std::vector<int> & values) {
    if (values.empty()) {
        throw std::invalid_argument{"empty..."};
    }
    sort(begin(values), end(values));
    return values[values.size() / 2];
}
```

### Correct

With an `std::vector` this example is correct. A vector provides random access iterators. Only random access iterators can be used with `std::sort` and they provide index access with `[]`.



- **The standard library provides the most needed data structures**
  - All work very similar with algorithms, as they have a similar API and provide iterators
  - Learn where to apply more efficient member functions instead of algorithms
    - E.g. `std::set::count()` member function vs. `std::count()` algorithm
- **Understand where to apply which data structure**
- **Iterators have different capabilities and provide corresponding member operators**